DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Sistema operativo rudimentario sobre arquitectura IA-32, con comportamiento/funciones de juego ambientado en la serie Rick and Morty

Organización del Computador II

Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Luschnat, Werner Alexis	356/08	w_luschnat@yahoo.com.ar
Herrera Gayol, Joaquín	273/17	joaco_hg@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

En este trabajo se implementa un sistema operativo rudimentario sobre arquitectura IA-32 usando código C y ASM para comportarse y tener funciones de juego ambientado en la serie $Rick\ and\ Morty$.

Índice

1. Introducción

Se implementa un sistema operativo básico sobre arquitectura IA-32. El mismo permite la ejecución de 24 tareas con privilegios de usuario que pueden llamar a rutinas del sistema (o “sysCalls”), más una tarea “Idle” (tarea “inútil” que el organizador de tareas a veces va a cargar). Un organizador de tareas (o “Scheduler”) define en qué ciclo de reloj corre qué tarea, que para esta implementación es en un orden fijo y predefinido.

Se utiliza un emulador de computadora compatible a IBM llamado Bochs, compilado con funciones de depuración (o debug).

Todo el sistema programado se compila sobre una imagen de floppy disc, y Bochs es configurado para bootear desde el mismo. Al inicial la emulación el BIOS copia a memoria los 512 bytes correspondientes al boot-sector provisto por la cátedra, de la dirección 0x7C00 en adelante y luego la ejecución continúa a partir de allí. Los pasos que ejecuta el boot-sector hasta que comienza la ejecución del kernel fueron implementados por la cátedra.

Las 24 tareas que sistema corre además de la tarea “Idle“ pueden ser de tipo “Rick”, “Morty” o “Cronenberg”, cada una con código específico.

2. Interpretación del juego

Los Ricks y Mortys pueden moverse sobre el tablero generando portales y atravesándolos. También pueden apuntar el portal sin atravesarlo, para conquistar Cronenbergs.

Para conquistar un Cronenberg, un Rick o un Morty tiene que apuntar su portal sobre el mismo, sin atravesarlo, pero modificando el código del Cronenberg. Esto hace que el Cronenberg sea conquistado. Cada vez que el Cronenberg se vuelve a ejecutar, dice “I am Rick C137” si lo conquistó Rick C137 o Morty C137, o dice “I am Rick D248” si lo conquistó Rick D248 o Morty D248.

En cada ciclo, la primera vez que la syscall IAmRick es ejecutada por un mismo Cronenberg conquistado, se incrementa el contador de códigos C137 (si se recibió C137) o el contador D248 (si se recibió D248).

Al final de la ronda, la interrupción de reloj escribe la cantidad recibida por cada uno en los cuadros de puntaje, y se reinician estos contadores para volver a contar en la siguiente ronda.

También puede suceder que un Rick o un Morty que se posiciona físicamente sobre el Cronenberg. En este caso el Cronenberg puede morir o no cuando toque su turno o en un turno subsiguiente. Va a ser desalojado y no ser vuelto a ejecutar. Un algoritmo del sistema que se ejecuta cuando es desalojado intenta deducir cuál equipo (si el C137 o el D248) fue el que causó la muerte para incrementar el puntaje del equipo.

El juego finaliza cuando alguno de los Ricks muere, declarándose el grupo contrario ganador.

3. Archivos de código

3.1. kernel - Arranque del sistema

El código del kernel es corrido una sola vez al inicio para poner en funcionamiento el sistema con sus funciones. A contunuación se enumeran los pasos específicos que realiza, uno tras otro, para finalmente pasarle el control al scheduler.

Se deshabilitan interrupciones con intrucción cli.
Se llama a la rutina A20_enable para habilitar la memoria más allá de 1MB.

Se carga la tabla de descriptores de segmento globales (o "GDT") con la instrucción lgdt y la dirección de memoria GDT_DESC (ésta definida en defines.h), en donde están preparados los descriptores de segmento.

Se enciende el bit menos significativo (o "PE", o "protect enable") del registro de control CR0, y se hace el salto al segmento de código de nivel 0, que correspone al índice 8 de la gdt, con offset a la siguiente línea de código del mismo kernel. Con ésto el procesador queda en modo protegido y todo el código que sigue compila compatible al modo 32 bit.

Se asigna el segmento de datos de nivel 0 a los registros selectores de segmentos de datos y fija la base de la pila en la dirección 0x27000.

Se carga el selector de segmento de video y se pinta en pantalla el área a utilizar como tablero, por única vez a través de este segmento.

Se inicializan variables del manejador de memoria llamando a la función mmu_init, que inicia el puntero a dirección de $proxima\_pagina\_libre$ en la llamada "área libre del kernel" (0x100000 hasta 0x3FFFFF), que contiene páginas libres que se irán utilizando para los mapas de memoria de las tareas, pero también para almacenar datos de otras funciones del kernel, como se describirá puntualmente más adelante.

Se genera y se carga un directorio y una tabla de páginas para mapear los primeros 4MB de memoria, que corresponden al kernel, con "identity mapping". Se llama a la función $mmu\_initKernelDir$ que devuelve sobre eax el puntero al directorio de páginas del kernel y éste se escribe en el registro de control cr3. Luego, se habilita paginación seteando en 1 el bit 32 del registro de control cr0.

Para probar que la paginación haya sido exitosa se escriben ambas libretas de los integrantes desde la dirección 0xB9EA0, que corresponde la memoria de video, verificándose la escritura en pantalla.

Se inicializa el organizador de tareas, o scheduler, con la función sched_init.

Se carga la tabla de descriptores de interrupciones (o "IDT") en memoria con la función idt_init. Se activa la IDT con la instrucción lidt a la que se le pasa por parámetro la dirección base de la IDT. Se configura el controlador de interrupciones con las funciones pic_reset y pic_enable. Se inicializa la tss con tss_init y se cargan las tareas de usuario con tss_user_tasks_init.

Finalmente, se carga el selector de segmento de la tarea inicial, se habilitan las interrupciones y se realiza el salto a la tarea "inútil" o "Idle". A partir de este momento las interrupciones de reloj harán que se ejecute la rutina "de atención de interrupciones de reloj", que es la que intercambia tareas procesador según el "agendador" o "scheduler", lo que se describirá más específicamente en las siguientes secciones.

3.2. pic - Configuración y mapeo de los 2 controladores de interrupciones PIC8259

Rutinas suministradas por la cátedra. Se mapean los canales de interrupciones en forma estándar.

3.3. gdt - Tabla de descriptores globales: Segmentación plana de código y datos. Descriptores de tss

3.3.1. Descriptores de segmentos de código-datos

2 descriptores definen 2 segmentos de código-datos de nivel 0, uno de datos y otro de código.

2 descriptores definen 2 segmentos de código-datos de nivel 3, uno de datos y otro de código.

Estos segmentos de código-datos se cargan al inicio del sistema sobre el procesador, para permanecer así y nunca ser alterados. Son "planos" (flat), es decir, todas las direcciones efectivas son traducidas al mismo valor de direcciones lineales. En adelante se llaman "virtuales" a las direcciones lineales, que son a su vez direcciones efectivas.

3.3.2. Descriptores de segmentos de estados de tareas (o descriptores de tss)

Los segmentos de estados de tareas (tss) de tarea inicial y tarea Idle son definidos tempranamente en gdt.c. Los descriptores correspondientes a tareas de usuario son cargados más adelante en tss.c.

3.4. idt - Tabla de descriptores de rutinas de atención de interrupciones (tanto de hardware como de software)

Se cargan descriptores para indicar el código a ejecutar de acuerdo al número de interrupción ingresado. Se utiliza una macro IDT_ENTRY suministrada por la cátedra para generar las entradas de interrupciones de nivel 0, y se genera una versión adicional IDT_ENTRY_3 para generar las entradas de las interrupciones 137, 138 y 139, de nivel 3.

3.4.1. Interrupciones activables solamente desde nivel 0

Interrupciones del procesador

0
...
31

Interrupciones externas

32 reloj
33 teclado

3.4.2. Interrupciones activables desde código de nivel 3, o "syscalls"

137 usePortalGun
138 iAmRick
139 whereIsMorty

3.5. isr - Rutinas de atención de las interrupciones (tanto de hardware como de software)

3.5.1. excepciones

Se utiliza y altera la macro ISR (suministrada por la cátedra) que genera el código para las rutinas de captura de excepciones. Se genera una versión adicional $ISR\_withErrorCode$.

La versión ISR de la macro, se utiliza para generar rutinas de atención para exepciones que NO generan código de error.

La versión $ISR\_withErrorCode$ de la macro, se utiliza para generar rutinas de atención para exepciones que NO generan código de error.

Las rutinas asignadas a las excepciones 0 hasta 31 llaman a $sched\_killCurrentTask$ para que la tarea que generó la excepción no sea vuelta a llamar.

Tienen en cuenta si el modo debug está activado ($debug\_mode = 0x00000DEB$). De ser así se llama a $screen_backup$, se levantan valores de registros de la pila y se los pasa a "screen_drawDebugBoxz el reloj mantendrá el sistema en Idle. Esto se detalla más ampliamente en la subsección modo debug.

Si el modo debug no está activado ($debug\_mode = 0x00000000$) solamente se imprime en la parte inferior de la pantalla la excepción ocurrida.

Se realiza un jump far a la siguiente tarea del scheduler, similar a como lo hace la rutina de reloj.

3.5.2. reloj

La rutina se fija si el sistema se encuentra en una parada de debug ($sched\_isStopped = 1$). De ser así no se realiza acción.

De NO encontrarse el sistema en una parada de debug, actualiza la aguja del deloj con $nextClock$.
Se resetean los contadores del juego correspondientes al turno actual (tick o ciclo de reloj actual) con llamando a $reset\_cicle\_counters$ (game.h, game.c).
Se obtiene la entrada de la gdt de la siguiente tarea con $sched\_nextTask$.
Se actualizan las agujas de reloj individual de la tarea a la que se saltará con $sched\_taskNextClock$.

Luego con un jump far se salta a la siguiente tarea.

3.5.3. teclado

La rutina de atención del teclado detecta la presión de la tecla “y” y procede según se detalla en la sección “debug mode”. De tratarse de otra tecla, llama a printNumTopRightEdge para, en caso de tratarse de teclas numéicas, imprimir en la esquina superior derecha de la pantalla el número.

3.5.4. syscalls

La rutina de atención 137 llama a la función usePortalGun_C.

La rutina de atención 138 llama a la función IamRick_C.

La rutina de atención 139 llama a la función IamRick_C.

3.6. mmu - Unidad de manejo de memoria

Contiene las rutinas encargadas de administar la memoria.
La variable global $proxima\_pagina\_libre$ indica la próxima página libre del kernel.

mmu_init inicializa $proxima\_pagina\_libre$ asignándole su valor inicial 0x100000.

$mmu\_nextFreeKernelPage$ devuelve el valor actual de $proxima\_pagina\_libre$ y luego deja el valor de esta variable global incrementado en 0x1000.

$mmu\_mapPage$ mapea una página virtual a una física, sobre un directorio pasado por parámetro y con atributos (salvo el presente que es verdadero siempre) pasados como parámetros.

mmu_unmapPage borra el mapeo virtual en la dirección pasada por parámetro.

$mmu\_initKernelDir$ realiza el mapeo identidad de los primeros 4MB de la memoria, devolviendo la dirección del directorio de páginas (0x27000). La dirección de la tabla de páginas afectada es 0x28000. esta función se utiliza en el kernel para cargar cr3 y activar por primera vez paginación.

$mmu\_mapTaskCodeAndStack$ realiza los mapeos de una página de código/datos y una página de la pila para una tarea de usuario sobre en dos páginas físicas contiguas a partir de una dirección física pasada por parámetro, llamando 2 veces a $mmu\_mapPage$. Las dos páginas virtuales también son contiguas y la primera es parámetro.

$mmu\_unmapTaskCodeAndStack$ se usa para desmapear las páginas de código y datos de una tarea. Se utiliza para deshacer un mapeo hecho con $mmu\_mapTaskCodeAndStack$, en particular se la llama desde usePortalGun como uno de los pasos para mover una tarea de una dirección física a otra.

$mmu\_initTaskDir$ mapea los primeros 4MB com papeo identidad y nivel 0 al igual que $mmu\_initKernelDir$, pero además llama a $mmu\_mapTaskCodeAndStack$ para mapear una página de código y una página de pila de nivel 3. Se utiliza desde tss.c cuando se inicializan las tss de cada tarea.

3.7. sched - Agendador de tareas

El scheduler administra el orden en el que se ejecutan las tareas.

$sched\_init$ inicia la variable global de tarea de usuario $actualsched\_task\_current$ en la última tarea de usuario $GDT\_IDX\_TSS\_USER\_LAST$ para que, al pasarse por primera vez a una tarea de usuario, se incremente y ejecute la primera tarea de usuario $GDT\_IDX\_TSS\_USER\_FIRST$.
$sched\_init$ también recorre el listado de tareas de usuario seteándolas como vivas con un $true$.

Si $game\_state.over == false$, la función $sched\_nextTask$ se encarga de devolver el índice de la GDT de la próxima tarea de usuario “viva” y actualiza la variable global $sched\_task\_current$. También llama a la función $game\_state\_set\_task\_type$ para actualizar la variable $taskType$ en la estructura $game\_state$. En caso de que $game\_state.over == true$, es decir el juego haya terminado, $sched\_nextTask$ devuelve el índice de la GDT de la tarea Idle.

3.8. screen - Interacción con BIOS para controlar la pantalla

Contiene las rutinas para administrar información en pantalla.

$screen\_backup\_init$ reserva dos páginas en el área libre del kernel para salvar la información de la pantalla antes de mostrarse la ventana de depuración.

$screen\_backup$ salva la pantalla previo a una ventana de debug y screen_restore la restaura a su estado anterior.

$screen\_drawDebugBox$ dibuja la pantalla de depuración y escribe los valores de los registros que recibe de parámetros en pantalla.

$screen\_showDebugModeLegend$ muestra la leyenda "DEBUG MODE" en el centro de la línea superior de la pantalla, y $screen\_hideDebugModeLegend$ la quita.
$print\_exception$ genera un pequeño "cartel" para indicar de una excepción ocurrida. Se utiliza para indicar excepciones en modo normal, modo en que la ejecución no se para y se desaloja la tarea que la produjo. Pero también se utiliza como título de la ventana de depuración.

$printNumTopRightEdge$ escribe en la esquina superior derecha el número (del 0 al 9) que es tecleado mientras el sistema está corriendo. Para determinar que número debe ser impreso, se utiliza la interrupción 33 de la isr que registra la tecla ingresada a través del puerto 0x60 correspondiente al teclado.

3.9. syscall - llamadas de las tareas al sistema implementadas como interrupciones

Suministrado por la cátedra.

Se otorga signatura C a las interrupciones 137, 138 o 139, que son interrupciones que las tareas que tienen nivel de usuario pueden invocar (y lo harán a través de las signaturas C).

3.10. game - Rutinas en C de las syscalls y estructuras de estado del juego

game.c contiene las funciones que se ejecutan luego de que una tarea llame a una syscall. También contiene una estructura llamada $game\_state$ que guarda información sobre variables que afectan al juego.

3.10.1. game_state

La estructura game_state contiene

usos de portal por Ricks que restan para que una Morty pueda usarlo,
$remaining\_calls\_by\_Rick\_before\_Morty$,

booleano para corroborar si ya fue creado un portal en el ciclo de reloj,
$portal\_called\_on\_cycle$,

variable que guarda código IamRick recibido de la tarea que está corriendo,
$I\_am\_Rick\_code\_received$,

booleano que indica si se recibió un código IamRick en el ciclo,
$I\_am\_Rick\_received$,

el tipo de tarea que está corriendo en el ciclo,
$taskType$,

contadores de Cronenbergs que enviaron IamRick C137, y Cronenbergs que enviaron IamRick D248,
$Cronenbergs\_C137\_count\_on\_cicle$,

$Cronenbergs\_D248\_count\_on\_cicle$,

contadores de Cronenbergs "asesinados" por cada equipo,
$Cronenbergs\_killed\_by\_C137$,
$Cronenbergs\_killed\_by\_D248$,

coordenadas de los últimos portales generados, sin ser atravesados (cross = false), por cada equipo,
$C137\_portal\_not\_crossed\_x$,
$C137\_portal\_not\_crossed\_y$,
$D248\_portal\_not\_crossed\_x$,
$D248\_portal\_not\_crossed\_y$,

booleano juego terminado (game over),
$over$.

3.10.2. reset_cicle_counters

Reinicia algunas variables de estado de juego
en cada ciclo de reloj:
- $portal\_called\_on\_cycle$ = FALSE
- $I\_am\_Rick\_code\_received$ = 0x0000
- $I\_am\_Rick\_received$ = FALSE

3.10.3. reset_round_counters

Actualiza en pantalla el puntaje de cada equipo
y luego reinicia los contadores de syscals IAmRick emitidos por Cronenbergs:
- $Cronenbergs\_C137\_count\_on\_cicle$ = 0
- $Cronenbergs\_C248\_count\_on\_cicle$ = 0

Es llamada desde la rutina de atención de interrupción de reloj, pero solamente cada vez que termina una ronda, y no en cada ciclo de reloj.

3.10.4. usePortalGun_C

$usePortalGun\_C$ es invocada por la interrupción vinculada a la $syscall\_usePortalGun$

$usePortalGun\_C$ se encarga de crear un portal apuntando a alguna posición del mapa relativa a la posición actual de la tarea Rick o Morty que invocó a $syscall\_usePortalGun$.

Primero se fija si fue creado ya un portal en el turno actual y cuántos usos de Rick faltan para que la tarea Morty pueda utilizarlo. Si los usos faltantes para el uso de Morty son 0, entonces Morty puede disparar.

A partir del puntero al directorio de páginas y estudiando el mapeo se calculan sus coordenadas de tablero x e y actuales de la tarea que invocó.

Si el valor de entrada $cross$ es $true$, se remapea el código de la tarea a la dirección física apuntada por el portalgun mediante $mmu\_RelocateTaskPhyscally$ que además se pisa el lugar de memoria anterior con código basura (para que produzca una excepción de haber sido el lugar físico de otra tarea). Se desplaza el cáracter de la memoria de video a la posición nueva y se pinta de verde la posición vieja.

Si además $withMorty$ vale $true$, se ejecuta $game\_calculateTaskCodepageAndWorldCoords$ para obtener las coordenadas x e y y la página física del Morty (que corresponde al Rick), para luego trasladar también físicamente (remapear) a esa tarea Morty, mediante $mmu\_RelocateTaskPhyscally$.

En caso que el valor de $cross$ sea $false$, se mapea la dirección física apuntda por el portal a la dirección virtual 0x08002000 y sobre esa dirección se escribe el código que llama a la syscall IamRick. Si había una tarea en la posición apuntada, cuando sea ejecutada dirá "I am Rick (código del conquistador)". Si no había tarea no sucederá nada, se escribió código en direcciones físicas que no serán ejecutadas, y se dibuja el portal en el mapa escribiendo un asterisco.

3.10.5. IamRick_C

$IamRick\_C$ es invocada por la interrupción vinculada a la $syscall\_IamRick$

La tarea que llama a esta función pasa un código (0xC137 o 0xD248). De ser el llamador una tarea Cronenberg, llama a la función $sched\_redrawTaskCondition$ para pintarlo de azul o rojo, según el valor pasado sea 0xC137 o 0xC248 respectivamente.

3.10.6. whereIsMorty_C

$whereIsMorty\_C$ es invocada por la interrupción vinculada a la $syscall\_whereIsMorty$

Estudia el mapeo de la tarea Morty vinculada al Rick, toma su dirección física y calcula sus coordenadas de tablero x y su y, para retornarlas sobre los punteros que recibió como parámetros.

3.10.7. sched_redrawTaskCondition

La función $sched\_redrawTaskCondition$ obtiene el puntero al directorio de páginas, a la tabla de páginas, la página de código y la página de la pila de la tarea actual, luego calcula las coordenadas x e y de la tarea para pintar de algún color de acuerdo a que jugador fue quien conquistó a esa tarea Cronenberg. Color gris para una tarea muerta, color azul para una tarea conquistada por el universo 0xC137 y color rojo para el universo 0xD248.

3.10.8. game_pointForKiller

$game\_pointForKiller$ intenta identificar cuál fue el equipo (0xC137 o 0xD248) que o bien introdujo código en la tarea que murió. Es llamada por $sched\_killCurrentTask$.

La función se fija primeramente si existe un Rick o un Morty que se haya desplazado sobre la posición física del código de la tarea. Esto lo hace comparando la página física de la tarea muerta con las páginas físicas de los Ricks y Mortys, que se obtienen usando $game\_calculateTaskCodepageAndWorldCoords$.

También se fija si un portal (de solamente 1 equipo y que no haya sido cruzado) tiene coodenadas

$C137\_portal\_not\_crossed\_x$ y $C137\_portal\_not\_crossed\_y$

o

$C137\_portal\_not\_crossed\_y$ $C137\_portal\_not\_crossed\_x$

coincidentes con la coordenadas x e y de la tarea (calculadas con $game\_calculateTaskCodepageAndWorldCoords$)

En caso de verificarse una de las situaciones de arriba se atribuye la muerte de la tarea a un equipo y se incrementa uno de los dos contadores $Cronenbergs\_killed\_by\_C137$ o $Cronenbergs\_killed\_by\_D248$.

Por último, si la tarea muerta es uno de los dos Rick, se informa en pantalla que el juego finalizó llamando a $screen\_drawGameOverBox$ a la que se le pasa el equipo ganador y el rick muerto, y se fija la variable $over$ en $true$. (La función $shed\_next\_task$ cuando sea llamada detectará que $over$ vale en $true$ y mantendrá el sistema en la tarea Idle.)

3.11. tss.c - Inicio de las tablas de estado de tareas (o tss)

Rutinas para cargar las tss de las tareas.

Se genera un arreglo de tss, se ingresan sus direcciones en la gdt.

$tss\_initial$ setea a todos los atributos/registros en la tss de la tarea inicial en 0.

tss_idle setea los atributos/registros en la tss de la tarea Idle. esp0, cr3, esp y ebp se setean en 0x27000 (misma pila del kernel), eflags en 0x202 para habilitar las interrupciones, el cs en la entrada de la gdt que contiene el descriptor de segmento de código de nivel 0, y el resto de los registros de segmento en la entrada que contiene el descriptor de segmento de datos de nivel 0.

$tss\_nextFreeTssTableIndex$ devuelve el próximo índice libre en la tabla o arreglo de tss mientras que $tss\_nextFreeGdtIndex$ devuelve el próximo indice libre en la gdt.

$tss\_user\_task\_init$ se encarga de completar una TSS con los datos de una tareas. Se crea el directorio de páginas con la función $mmu\_initTaskDir$. Se copian 8kb (dos páginas) desde la dirección src_base_addr hacia la dirección $task\_base\_virtual\_address$, entre otros seteos.

$tss\_user\_tasks\_init$ se encarga de cargar las tareas Rick, Morty y Cronenberg llamando a $tss\_user\_task\_init$ con parámetros particulares, usándose "posiciones aleatorias" suministradas por la cátedra dentro del mapa del mundo Cronenberg. Se marca en pantalla cada una de las tareas.

3.12. debug mode and window - Modo y ventana de depuración

El sistema permite un modo de depuración (o debug mode). De estar activado y ocurrir una excepción, las rutinas de atención de interrupciones activan una parada de depuración (debug stop).

La etiqueta $debug\_mode$ (en isr.asm) guarda
0x00000000 si modo debug está apagado, y
0x00000DEB si modo debug está encendido.

La variable global $debug\_stop$ (en sched.c) guarda
0x00 si NO se está y
0x01 si SÍ se está en una parada de depuración.

Cuando la presión de la tecla "y" es detectada por la rutina de atención de ints. de teclado, se llama a $sched\_isStopped$ para leer el valor de $debug\_stop$. De no estar en una parada de depuración, se activa o desactiva el modo debug. De estar en una parada, para salir de la misma y volver a la "normalidad" se llama a screen_restore y a $sched\_unsetDebugStop$, que setea en 0x00 la variable global $debug\_stop$.

Cuando se ejecuta alguna de las rutinas de atención de excepciones, las mismas inmediatamente ejecutan $pushad$ para salvar todos los registros posibles sobre la pila, verifican si $debug\_mode$ vale 0x00000DEB y de ser así activan $debug\_stop$ y llaman a $screen\_drawDebugBox$ para mostrar la pantalla de depuración.

La pantalla de depuración recibe por parámetros todos los valores de registros a mostrar en pantalla, salvo los de los selectores de segmento, salvo los últimos valores de la pila de nivel 3 que son escritos luego, desde el mismo código assembler de la rutina de atención.

La rutina de interrupción de reloj a su comienzo lee $debug\_stop$. Si vale 0x01, no realiza ningún intercambio de tarea porque el sistema debe mantenerse en Idle.

3.13. Juego terminado (o "game over")

El juego termina solamente si un Rick muere, y en ese caso el otro equipo gana. No se implementa corte de juego pasada una cierta cantidad de ciclos sin que un Rick muera, es decir, un Rick tiene que morir para que termine el juego.

Se informa en pantalla el equipo ganador (el del Rick que no murió) y también se informa el código del Rick que murió.